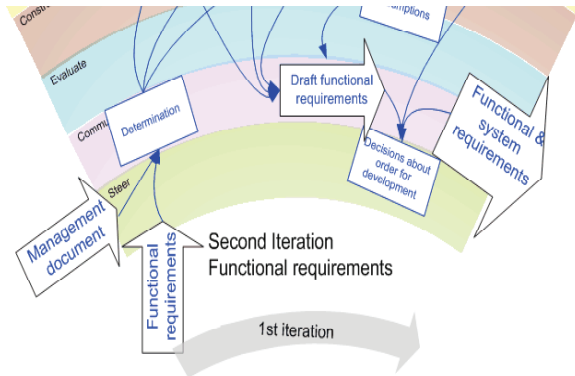


## ITERATION 2: FUNCTIONAL REQUIREMENTS

### STORIES AND PLANNING GAME



Bite: Functional requirements are further defined
Time: 2 hours
Input: Notes from client discussions, research notes, ERD.
Process evidence: Draft list of functional requirements.
Client: Discussion

In Iteration One, we used the Planning Game to identify some high level requirements for the system. These requirements were then used to define the scope of the system (metaphor), for the ethical analysis, and were the basis for the first release. They have provided a communication tool for discussing the project with the client and within the development group.

The agile movement's approach to requirements is two-fold

1. Requirements always change
2. It is not possible to define all requirements at the beginning of the project

It is important to understand that requirements are not a magical, defined entity. No-one knows what all the requirements are. Each user will have a perspective, and all views may be correct. We have described the requirements determination stage as a 'fishing expedition'. Cohn (2005) extends this metaphor by explaining that "different sized nets can be used to capture different sized requirements" (p43). Our initial Planning Game was the first pass, capturing the biggest and most obvious requirements. In this iteration we will use a smaller mesh size, to 'capture' the smaller requirements.

Further, Cohn explains that "requirements, like fish, mature and possibly die" (p43). Requirements defined in the first iteration may become redundant as other requirements are defined, or what seemed a minor requirement may become a critical component of the project. (These are the requirements that begin as a "wouldn't it be good if the system could also...?" conversation.)

And lastly on the fishing theme, while you won't capture all the requirements on your trawling expedition, and you might haul in some extraneous seaweed or crabs, a skilled requirements trawler will use the best techniques to make the expedition efficient.

### Requirements Specification Documents

Traditional software development methodologies use Requirements Specification documents to define the scope of a project. This approach assumes that we can "get the requirements right the first time, complete, concise and clear.." [http://satc.gsfc.nasa.gov/support/STC\\_APR98/do\\_reqmnt/do\\_reqmnt.pdf](http://satc.gsfc.nasa.gov/support/STC_APR98/do_reqmnt/do_reqmnt.pdf)

Many large development organisations publish requirements standards, which suppliers need to comply with for tender purposes. Check out the NASA website for a great source of these. A requirements specification document can be huge and defines the requirements down to the smallest detail. Many months and sometimes years can be dedicated to this level of requirements analysis.

As an example of the level of detail involved, the requirements specification for the ASPERA-3 Processing and Archiving Facility for the Mars Lander includes this requirement:

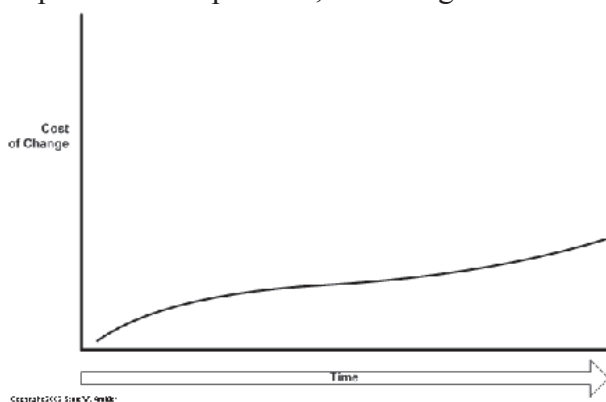
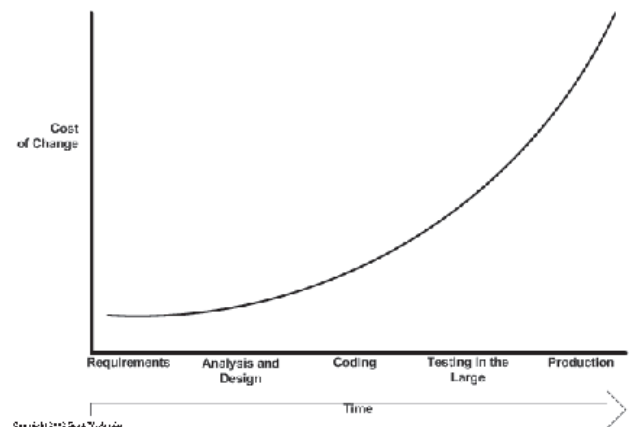
“ASPERA-3 and MEX OA IDFS data and any APAF-generated intermediate files of ASPERA-3 and MEX OA cleaned-up telemetry shall be provided to all ASPERA-3 Co-I’s.” (from [www.aspera-3.org/idfs/APAF\\_SRS\\_V1.0.pdf](http://www.aspera-3.org/idfs/APAF_SRS_V1.0.pdf))

These documents are a result of a structured, often waterfall, methodology. They are a reasonable attempt to bring some certainty into the software development process, in the context of rampant project failure. They also provide a legal and contractual basis for the development. The financial justification for this approach stems from a view that change in a project is a bad thing and that as soon as we can agree on the “correct” requirements we can move on to the design and build stages.

## Cost of Change Graphs

The Cost of Change curve is hotly debated in the Agile forum.

The traditional view (here we are talking about a single release of a product in a waterfall methodology) holds that changes in requirements occurring in the early stages of the development cycle are cheap to implement because little has been invested in the project. A change in requirements during the design stage will involve reworking of analysis and design, and will cost more. Once programming begins, cost start to mount and organisations are likely to resist the work required to repair the damage. (“It will be OK, we can add that feature in the next release”). After the product has been released, changes are prohibitively expensive to implement, involving new releases of the software, updated user documentation and damage to the developer’s reputation.



The agile version suggests that this problem of the cost of change can be resolved. The flatter agile curve is produced by ensuring that feedback loops are shortened and that every requirement (user story) is tested and verified before it is implemented. Any incremental or iterative methodology will improve the change curve, by preventing investment in faulty code.

## Rule No1 for requirements. Requirements will change.

In keeping with the agile principles of Extreme Programming, we need to embrace change and use communication with the stakeholders to ensure that our understanding of the requirements is sufficient for the development process. Boehm and Turner (2004) state that “successful, sustainable software development requires both discipline and agility” (p23). It is acknowledged that the agile methods are best suited to smaller projects where the client is accessible and requirements may change. While the overhead of a 200 page require-

ments document is not necessary for most projects, it can be useful to record the requirements to ensure clear understanding has been agreed with the client, especially in larger or mission critical projects. Writing good requirements is a useful skill which will advance the development team's understanding of the project.

“Agilists create high-value documentation” (Ambler, 2006). We will use the planning game to elicit a further set of deeper, more detailed requirements. In this second iteration we need to understand the user perspective – what does the user need the system to do? We will produce a draft requirements document that will become the basis for your development through the second iteration. Have the courage to change your requirements as your understanding of the project changes. Requirements may be deleted, added or edited at any time during the development. (You will need to justify these decisions!)

## WRITING GOOD REQUIREMENTS.

### Six qualities of good requirements

1. **Necessary** The stated requirement is an essential capability, physical characteristic, or quality factor of the product or process. If it is removed or deleted, a deficiency will exist, which cannot be fulfilled by other capabilities of the product or process.
2. **Verifiable** The stated requirement is not vague or general but is quantified in a manner that can be verified by one of inspection, analysis, demonstration or test.
3. **Attainable** The stated requirement can be achieved at a definable cost. Technically feasible, fits within budget, fits schedule,
4. **Clear** Each requirement expresses a single thought. It is concise and simple, easy to read and understand.
5. **Complete** Each requirement is standalone, no additional explanation should be necessary.
6. **Consistent** The stated requirement does not contradict other requirements or duplicate another requirement.

## Common Problems

### Language

The same term is used for the same item in all requirements.

Each requirement can usually be written in the format:

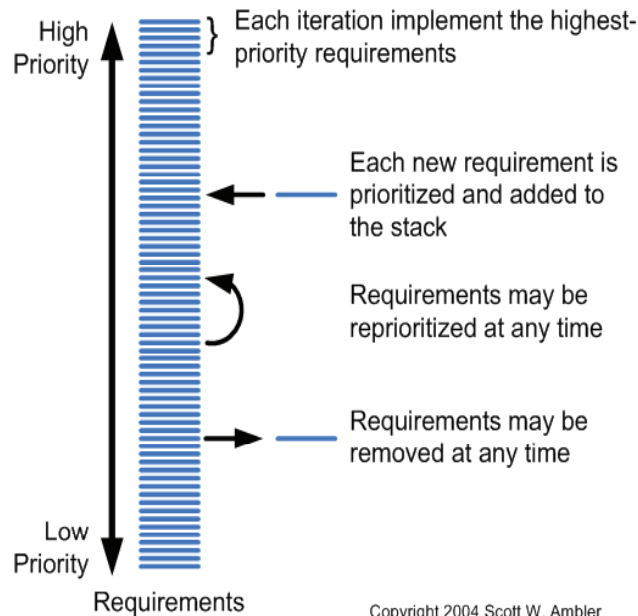
The System shall provide .....

The System shall be capable of .....

(It is the industry convention to use “Shall” for requirements, “Will” for statements of fact and “Should” for goals.)

The language of the requirements is important. If you find yourself using “Are”, “is”, “was” you are probably describing the system rather than its functionality. Eg, *“The system is storing the student data”*. Preferable is *“the system shall store student data”*

Avoid unnecessary words such as “must”, “support” (as in “system will support backups”), “but not limited to”, “etc”, “and/or” and lists. They cannot easily be defined, cause uncertainty and leave room for multiple interpretations. If requirement is not definable, it may need to go back to the user for further clarification.



Avoid weak phrases such as: Minimize / Maximize, Rapid, User-friendly, Easy, Sufficient, Adequate, Quick.

It is preferable to define the actual limits of the system. Eg, rather than “the system shall enrol students quickly” use “the system shall process each student enrolment within 24 hours.”

If insufficient information is available – poor requirements will be written because developers will make assumptions about missing information. The solution is to communicate frequently and clearly.

Avoid describing implementation in requirements. You need to focus on the what, not the how Eg, “provide a database” Preferable: “provide capability for storing, displaying and sorting customer data”

This may be a database, but allows for other possibilities, eg an xml file.

Avoid describing the use of the system.

Eg, “System shall allow user to open a dialogue box and enter data”

Preferable: “system shall allow data entry”

### Prioritising

First set of requirements/user stories is often a wish list, including many ‘nice to have’ features.

Refinements and clarifications are negotiated during the course of a project, and the requirements/user stories are defined in further iterations. Additional requirements or changes to existing requirements are identified in the normal course of engineering a solution.

Who should be involved in prioritising the requirements/stories?

Client

Developers

Stakeholders

A traditional approach divides requirements into

“Essential” requirements are those that must be included in the system

“Useful” requirements are those that would reduce system effectiveness if left out

“Desirable” requirements are those that are not part of the core, but make the system more attractive to the users priorities.

The agile approach uses the successive iterations to identify the most critical requirements. Which requirements or tasks will be implemented in this iteration? the next iteration? Both Scrum and eXtreme Programming approaches use a requirements stack to prioritise requirements. The story cards are ranked in order, with new cards added at the agreed level. Cards can be removed or reprioritized. At any time, the developers are implementing the highest priority requirements.

## SUMMARY

Writing good requirements and user stories is difficult, requires careful thinking and analysis, but is not magical. With experience, developers can improve the quality of the project and minimise reworking by writing effective and realistic requirements.

## REFERENCES

Cohn, M. User Stories

Boehm, B and Turner, R. Balancing Agility and Discipline

<http://www.agilemodeling.com/essays/changeManagement.htm>

[http://satc.gsfc.nasa.gov/support/STC\\_APR97/write/writert.html](http://satc.gsfc.nasa.gov/support/STC_APR97/write/writert.html)

<http://www.agilemodeling.com/essays/costOfChange.htm>

<http://www.complianceautomation.com/papers/writingreqs.htm>

<http://www.complianceautomation.com/papers/ManagingRequirements.pdf>

[http://www.complianceautomation.com/papers/incose\\_goodreqs.pdf](http://www.complianceautomation.com/papers/incose_goodreqs.pdf)

<http://www.complianceautomation.com/papers/ACaseforPriority.pdf>

